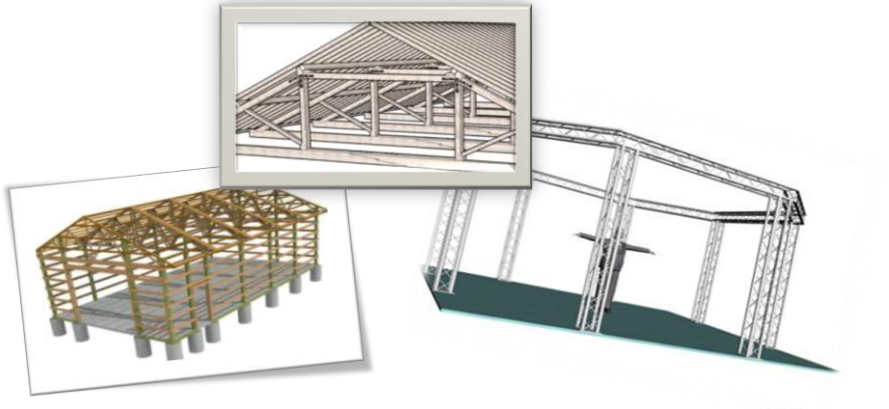
Gof Design Patterns

STRUCTURAL PATTERNS

1



Structural Patterns

Concerned with the structural relationships between classes and objects

Facade

Façade Meaning

4

- fa·cade –
- The face of a building, especially the principal face.
- An artificial or deceptive front: *ideological slogans that were a façade for geopolitical power struggles.*

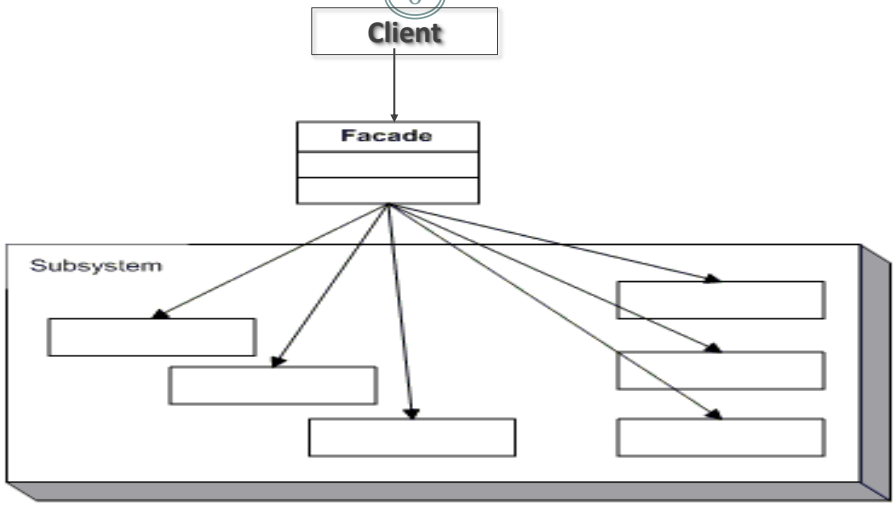
Façade Pattern

Make a complex system simpler by providing a unified or general interface, which is a higher layer to these subsystems

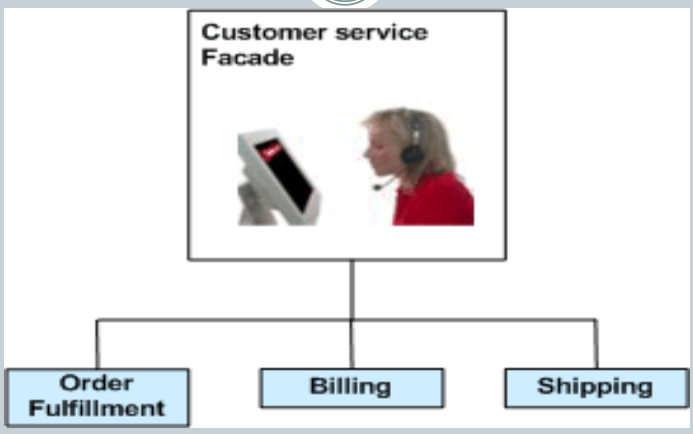
Benefits of Facade Pattern

- Want to reduce complexities of a system.
- Decouple subsystems , reduce its dependency, and improve portability.
- Make an entry point to your subsystems.
- Minimize the communication and dependency between subsystems.
- Security and performance consideration.
- Shield clients from subsystem components.
- Simplify generosity to specification.

Structure



Example



Illustrated Code

```
public class OrderCompletionFacade {
    /**
     * facade method takes in Order details and does the following 1. Process
     * the order completion by calling the order module 2. Updates shipping
     * information 3. Bills the order and Updates billing information
     */
    public void completeOrder(String orderId, String paymentId) {
        new OrderProcessing().completeOrder(); // might fetch and pass Order object to this method

        new BillingImpl().processBilling(); // might pass billing object to this method

        new Shipping().initiateShippingForOrder(); // might create a new shipping object and pass
    }
```

```
class BillingImpl {
    public void processBilling() {
        // contains logic for processing the billing given the credit card
        // details
    }
}
```

```
class OrderProcessing {
    public void completeOrder() {
        // Updates the order records and completes the order
```

```
class Shipping {
    public void initiateShippingForOrder() {
        // initiates the shipping request
    }
    public void captureShippingDetails() {
        // captures the address and does basic verification for zip code etc.
    }
}
```

Façade

9

- Mandatory v/s Optional Façade
- Web Services and service-oriented architecture provide a Façade for an entire system

Façade Pattern

Adapter

Adapter

- Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. (Also known as Wrapper)

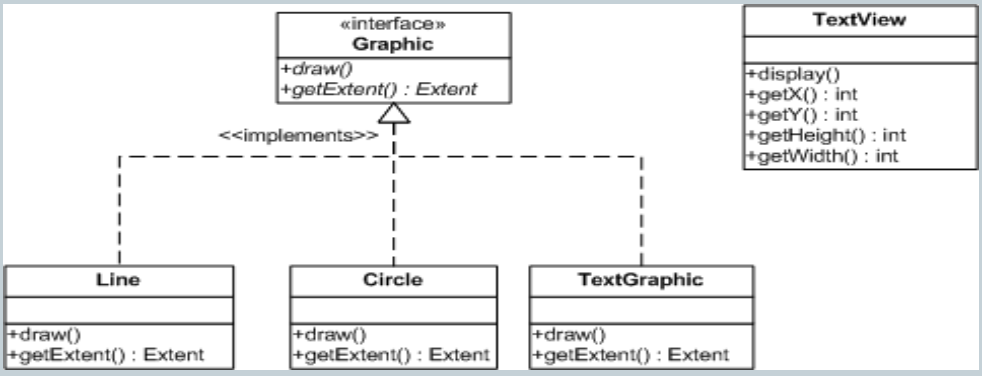
Benefits of Adapter Pattern

- Make unrelated classes work together.
- Multiple compatibility.
- Increase transparency of classes
- Make a pluggable kit.
- Increases class reusability
- Achieve the goal by inheritance or by composition

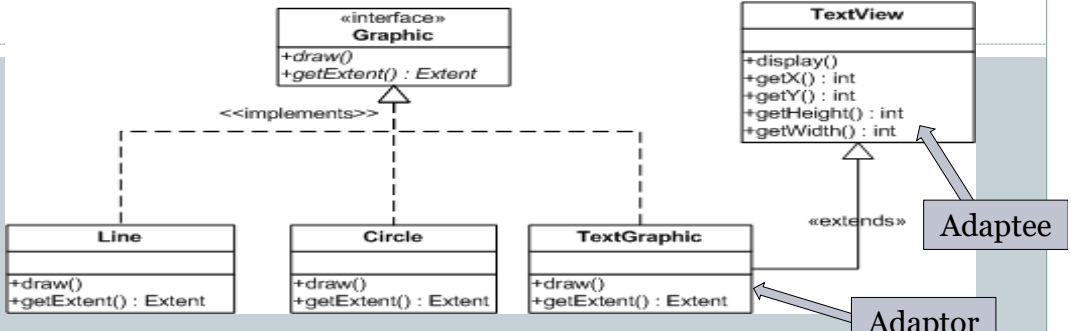
Adapter Pattern

- Class Adapter (Inheritance)
- Object Adapter (Composition)

Example

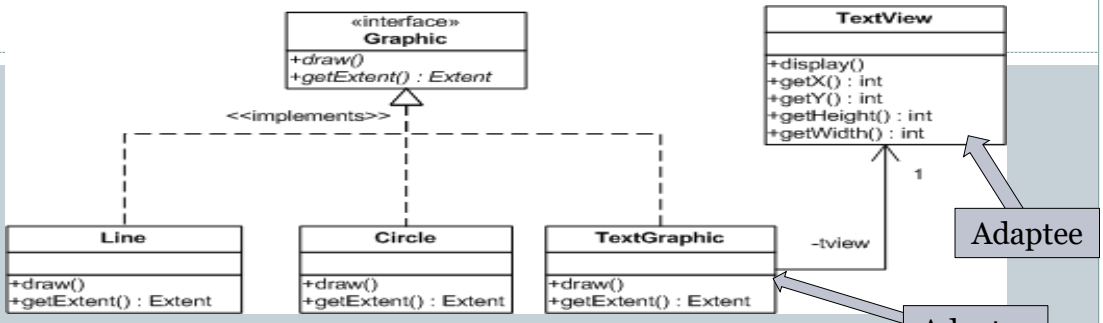


Class Adapter




```
public class TextGraphic extends TextView implements Graphic
{
   public void draw()
   {
        display();
   }
   public Extent getExtent()
   {
        return new Extent(getX(),getY(),getHeight(),getWidth();
        . . . .
   }
}
```

Object Adapter

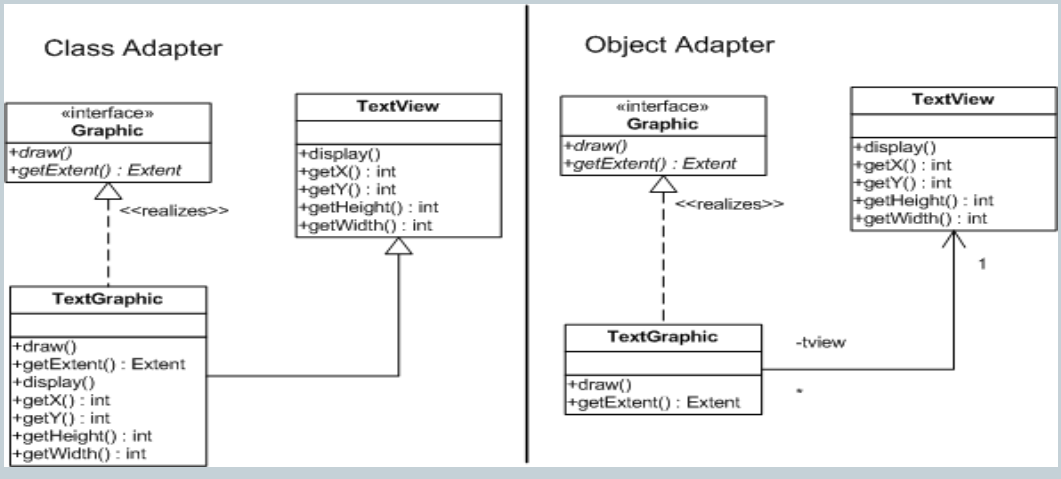



```
public class TextGraphic implements Graphic
{
   private TextView tView;
   public void draw()
   {
        tview.display();
   }
   public Extent getExtent()
   {
        return new Extent(tView.getX(), tView.getY(), tView.getHeight(),
                  tView.getWidth();
        . . . .
   }
}
```

Adapter Pattern

Adapter

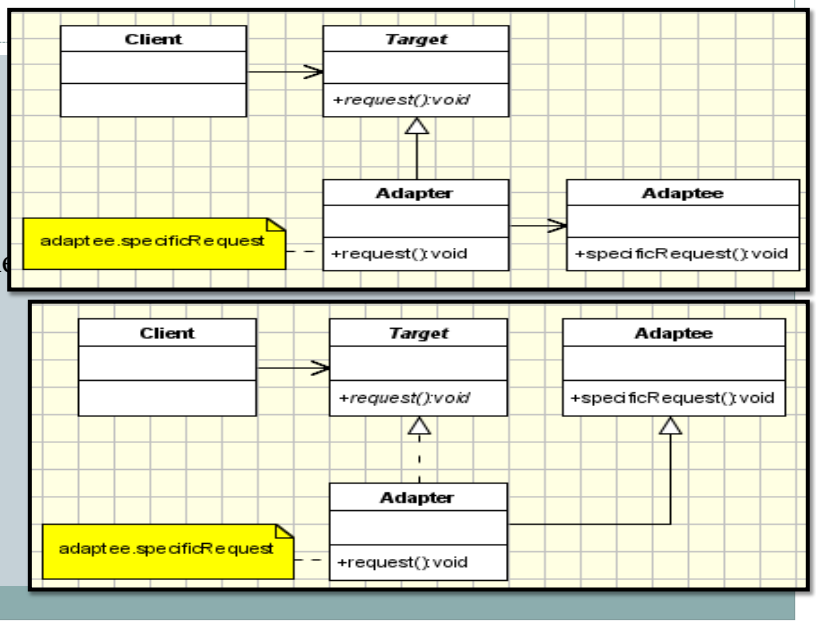
16



Adapter Pattern

Structure

Target - defines the domain-specific interface that Client uses.
Adapter - adapts the interface Adaptee to the Target interface.
Adaptee - defines an existing interface that needs adapting.
Client - collaborates with objects conforming to the Target interface.



Adapter Classes in Java & C#

- Java
  - javax.print.event.PrintJobAdapter
  - Event adapters :
    - java.awt.event.ComponentAdapter,
    - java.awt.event.MouseAdapter,
    - java.awt.event.ContainerAdapter,
    - java.awt.event.KeyAdapter ,
    - java.awt.event.MouseInputAdapter
  - org.xml.sax.helpers.XMLReaderAdapter (Adapt a SAX2 XMLReader as a SAX1 Parser.)
  - org.xml.sax.helpers.ParserAdapter (This class wraps a SAX1 Parser and makes it act as a SAX2 XMLReader, with feature, property, and Namespace support.)
- C#
  - System.Data.Common.DataAdapter class used with various data sources such as OleDB, Sql, and Oracle
  - Reader/Writer classes are adapters which convert a byte array interface to string/char interface.

Adapter Pattern

- **Applicability**
  - Use the Adapter pattern when you want to use an existing class, and its interface does not match the one you need
  - You want to create a reusable class that cooperates with unrelated classes with incompatible interfaces
- **Implementation Issues**
  - How much adapting should be done?
    - Simple interface conversion that just changes operation names and order of arguments
    - Totally different set of operations
  - Does the adapter provide two-way transparency?
    - A *two-way adapter supports both the Target and the Adaptee interface. It* allows an adapted object (Adapter) to appear as an Adaptee object or a Target object

Adapter Pattern

Bridge

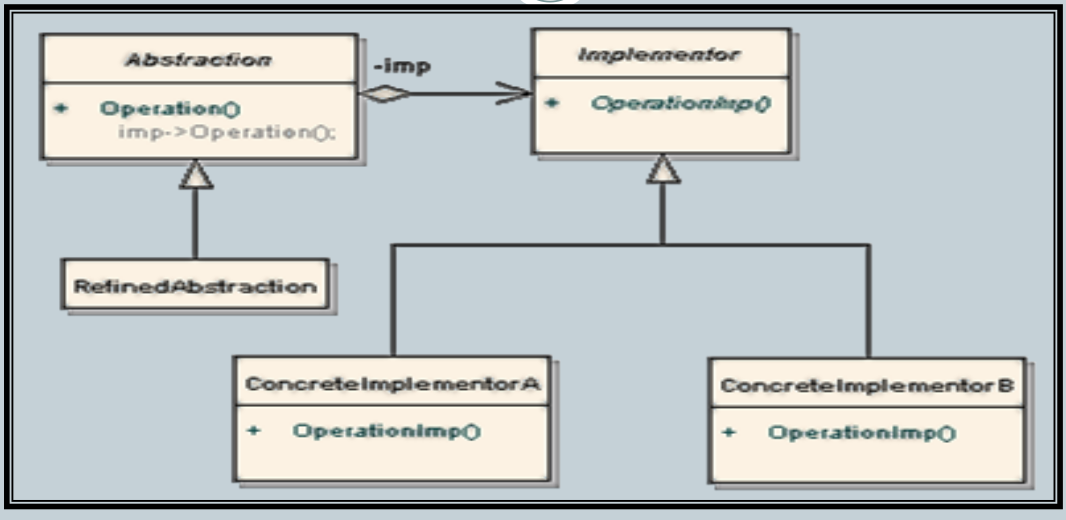
Bridge Pattern

Decouple an abstraction from its implementation so that the two can vary independently.

Usage

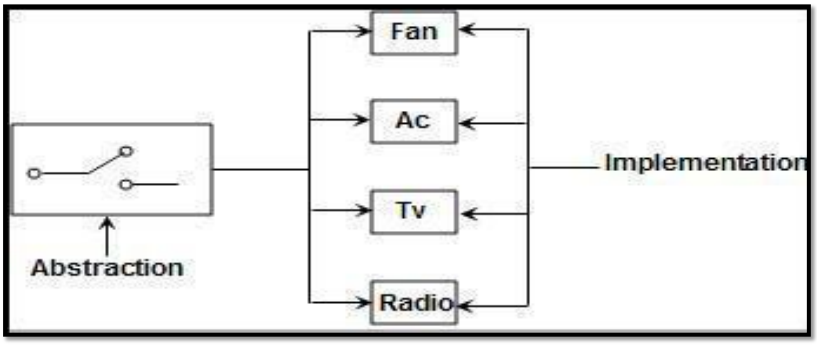
- When you want to avoid a permanent binding between an abstraction and its implementation.
- When both the abstraction and implementations should be extensible using subclasses.
- When changes in the implementation of an abstraction should have no impact on clients, which means that you should not have to recompile their code.

Structure



Simple Example

23



Bridge pattern between interfaces – IEquipment & ISwitch

```
public interface IEquipment
{
    void Start();
    void Stop();
}
```

Inheritance

```
class clsRefrigerator : IEquipment
{
    public  void Start()
    {
        Console.WriteLine("Start Compressor");
        Console.WriteLine("Start Ice Cooling");
    }
    public  void Stop()
    {
        Console.WriteLine("Stop Ice Cooling");
        Console.WriteLine("Stop Compressor");
    }
}
```

```
class clsBulb : IEquipment
{
    public void Start()
    {
        Console.WriteLine("Warm up the bulb");
```

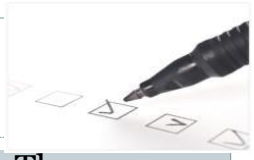
```
interface ISwitch
{
    void On();
    void Off();
}
class Clsswitch : ISwitch
{
    private IEquipment objEquipment;
    public void setEquipMent(IEquipment Equipment)
    {
        objEquipment = Equipment;
    }
    public void On()
    {
        objEquipment.Start();
    }
    public void Off()
    {
        objEquipment.Stop();
    }
```

Abstraction interface

Aggregated Equipment object reference

Implement both the interface

How to use Bridge pattern



25

- Decide if two orthogonal dimensions exist in the domain. These independent concepts could be: abstraction/platform, or domain/infrastructure, or front-end/back-end, or interface/implementation.
- Design the separation of concerns: what does the client want, and what do the platforms provide.
- Design a platform-oriented interface that is minimal, necessary, and sufficient. Its goal is to decouple the abstraction from the platform.
- Define a derived class of that interface for each platform.
- Create the abstraction base class that "has a" platform object and delegates the platform-oriented functionality to it.
- Define specializations of the abstraction class if desired.

Bridge Pattern in Java & C#

26

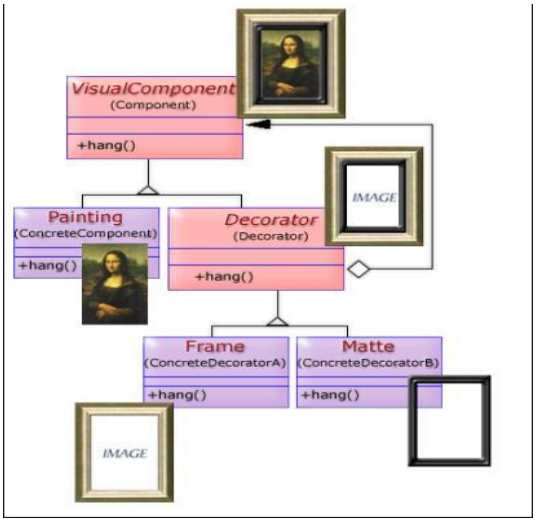
- Bridge pattern is used to separate Components and Component Peers.
- Persistence frameworks also use this pattern commonly

Consequences

27

- Decoupling interface and implementation. An implementation is not bound permanently to an interface. The implementation of an abstraction can be configured and even switched at run-time.
- Abstraction and Implementor hierarchies can be extended independently.
- Improved extensibility

Bridge Pattern

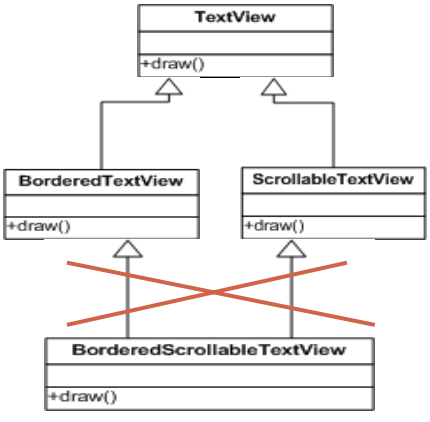


Decorator

Decorator

- Provide additional functionality to a class without subclassing it
- Provide additional responsibilities to an object dynamically

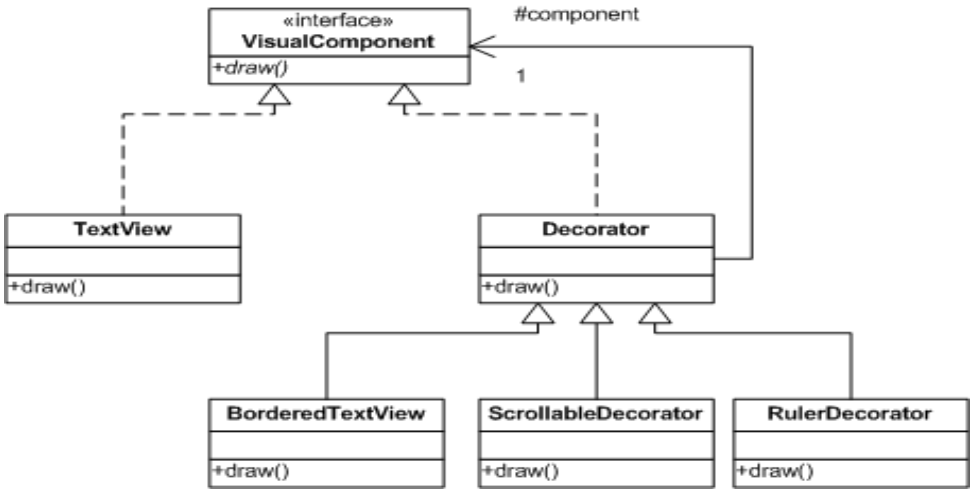
Problem statement



Now consider providing a ruler also for a TextView.

Solution using Decorator Pattern

31



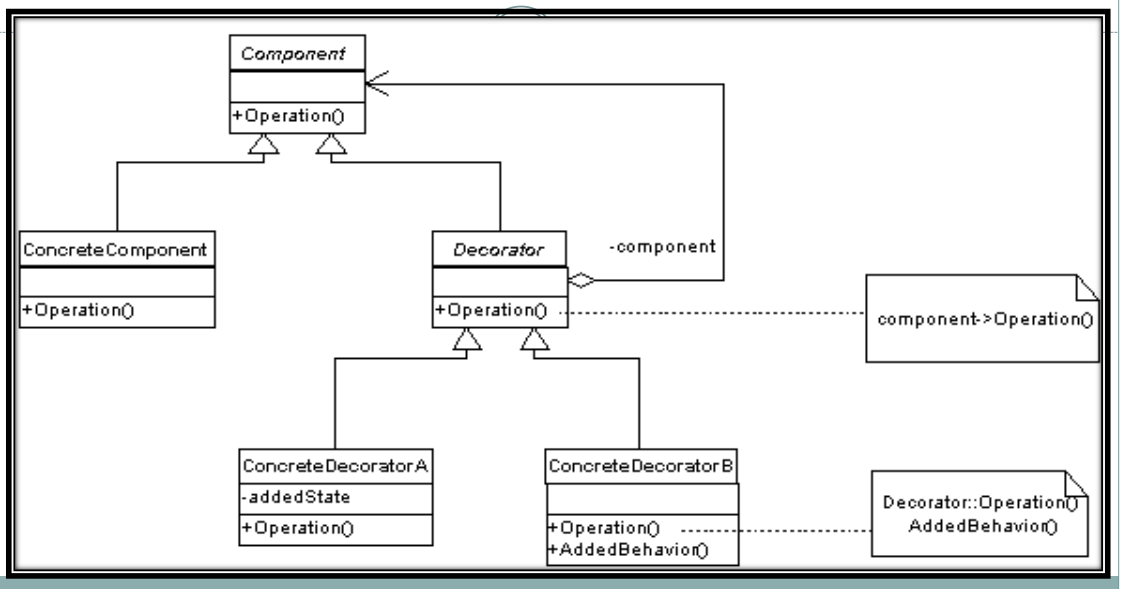
Decorator

```
public abstract class Decorator implements VisualComponent
{    // protected member variable
     protected VisualComponent component;
     // …
}
public class BorderDecorator extends Decorator{
     public BorderDecorator(VisualComponent comp){
          component = comp;
     }
     public void draw() {
          component.draw();
          drawBorder();
     }
     private void drawBorder() {
          //…
     }
}
```

```
public class ScrollbarDecorator extends Decorator{
     public ScrollbarDecorator(VisualComponent
     comp){
          component = comp;
     }
     public void draw() {
          component.draw();
          drawScrollbar();
     }
     private void drawScrollbar() {
```

```
// simple Text View
VisualComponent vc1 = new TextView();
vc1.draw
VisualComponent vc2 = new BorderDecorator(vc1);
vc2.draw
VisualComponent vc3 = new BorderDecorator(new ScrollbarDecorator(vc1));
```

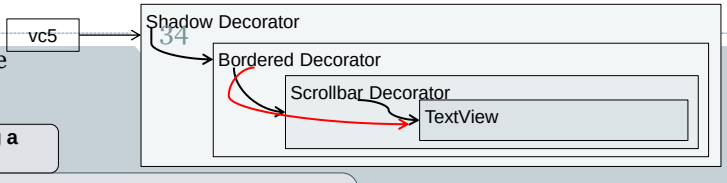
Structure



Decorator Pattern

Decorator Pattern Explained

Many small objects are created



Dynamically removing a Decorator

```
// if client wants to remove scrollbar
vcTemp = vc3;
while(vcTemp instanceof Decorator {
    if(vcTemp  instanceof  ScrollbarDecorator) {
            vcTemp.setComponent =
            vcTemp.getComponent();
            break;
    } else {
            vcTemp = vcTemp.getVisualComponent();
    }
}
```

1 : vcTemp → ShadowDecorator

2 : vcTemp → BorderedDecorator

3 : vcTemp → ScrollbarDecorator
BorderedDecorator.component =
ScrollbarDecorator.component

Decorator in Java & C#

35

- In Java
  - All subclasses of java.io.InputStream, OutputStream, Reader and Writer have a constructor taking an instance of same type.
  - Almost all implementations of java.util.List, Set and Map have a constructor taking an instance of same type.
- In C#
  - System.IO.Stream
  - System.IO.BufferedStream
  - System.IO.FileStream
  - System.IO.MemoryStream
  - System.Net.Sockets.NetworkStream
  - System.Security.Cryptography.CryptoStream

Applicability

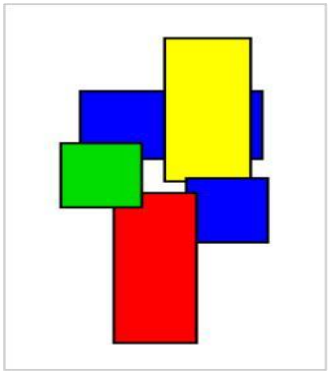
36

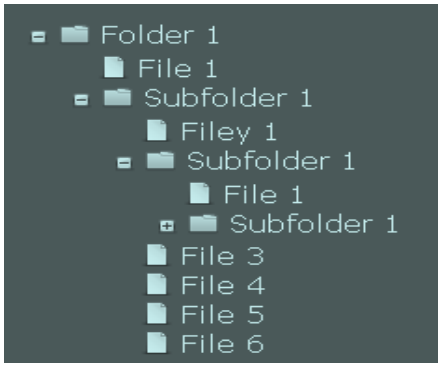
Use the Decorator pattern

- To provide various additional functionalities to an existing class without sub-classing and when the functionalities are more decorative in nature. If a large number of independent extensions and their combinations are possible, subclassing may result in an explosion of classes
- To add responsibilities to individual objects dynamically and transparently, without affecting other objects
- When a class to be extended may not be available for subclassing (final classes in Java)

Consequences

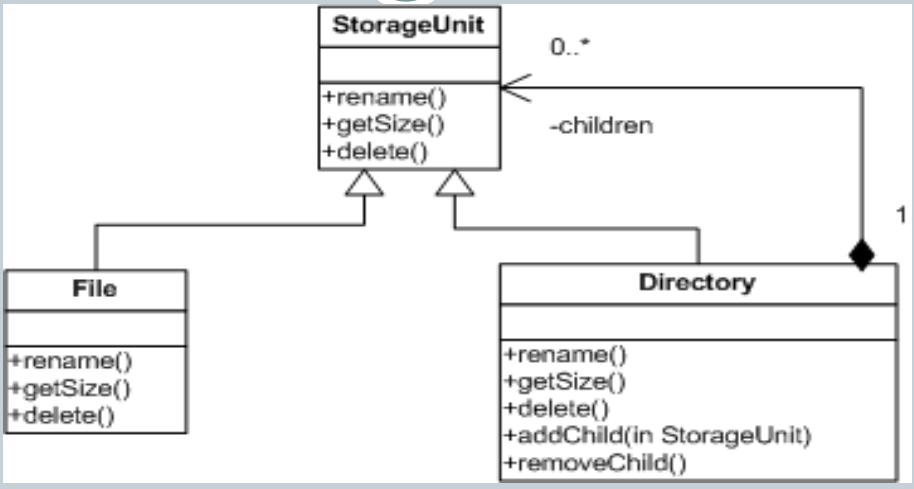
- More flexible than inheritance. Responsibilities can be added / removed dynamically.
- A property can be added multiple times, if required
- Avoids feature-heavy classes high up in the hierarchy. Clients use objects on a pay-as-you-use basis.
- A decorator and its component are not identical
- Results in many small objects





Composite

Windows Explorer example



Windows Explorer example illustrated with code


```
interface StorageUnit {

    public void rename(String name);
    public int getSize();
    public boolean delete();
    public String getName();

}
```

Component

```
class File implements StorageUnit {
    private int size;
    private String name;
    public File(String name, int size){
        this.size = size;
        this.name = name;
    }
    @Override
    public boolean delete() {

        return false;
    }
    @Override
    public int getSize() {
        return size;
    }
    @Override
    public void rename(String name) {
        this.name = name;
    }
    @Override
    public String getName() {
        return this.name;
    }
}
```

Leaf

Code contd..

Composite class

```
class Directory implements StorageUnit {
    //composite of other Storage units
    protected List<StorageUnit> childStorageUnits =
        new ArrayList<StorageUnit>();
    private String folderName;

    public Directory(String folderName)
    {
        this.folderName = folderName;
    }
```

Simple overridden functions

```
    @Override
    public void rename(String folderName) {
        this.folderName = folderName;
    }
    @Override
    public String getName() {
        return folderName;
    }
```

Child management functions

```
    public void add(StorageUnit childStorageUnit) {
        childStorageUnits.add(childStorageUnit);
    }
    public void delete(StorageUnit childStorageUnit) {
        childStorageUnits.remove(childStorageUnit);
    }
```

Composite Pattern

Code contd..

```
class Directory implements StorageUnit {
    //composite of other Storage units
    protected List<StorageUnit> childStorageUnits =
        new ArrayList<StorageUnit>();
```

Composite methods

```
    @Override
    public boolean delete() {
        for (StorageUnit childStorageUnit : childStorageUnits) {
            childStorageUnit.delete();
        }
        deleteFolder(); // delete the folder itself
        return true;
    }
```

```
    @Override
    public int getSize() {
        int foldersize =0;
        for (StorageUnit childStorageUnit : childStorageUnits) {
            int childStorageUnitSize = childStorageUnit.getSize();
            foldersize += childStorageUnitSize;
        }
        System.out.println(this.folderName +" - "+ foldersize);
        return foldersize;
    }
```

Composite Pattern

Composite

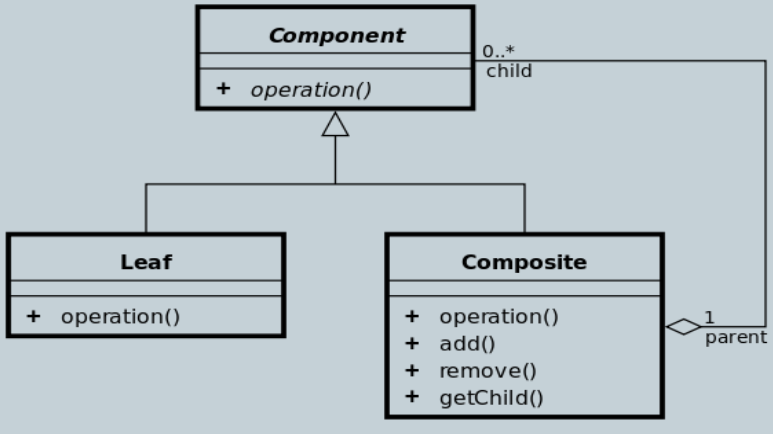
43

- Used for whole-part structures (aggregation hierarchies)
- To provide polymorphic access to whole objects and part objects
- Group components to form larger components, which in turn can be grouped to form still larger components.
- Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- Recursive composition

Composite Pattern

Structure

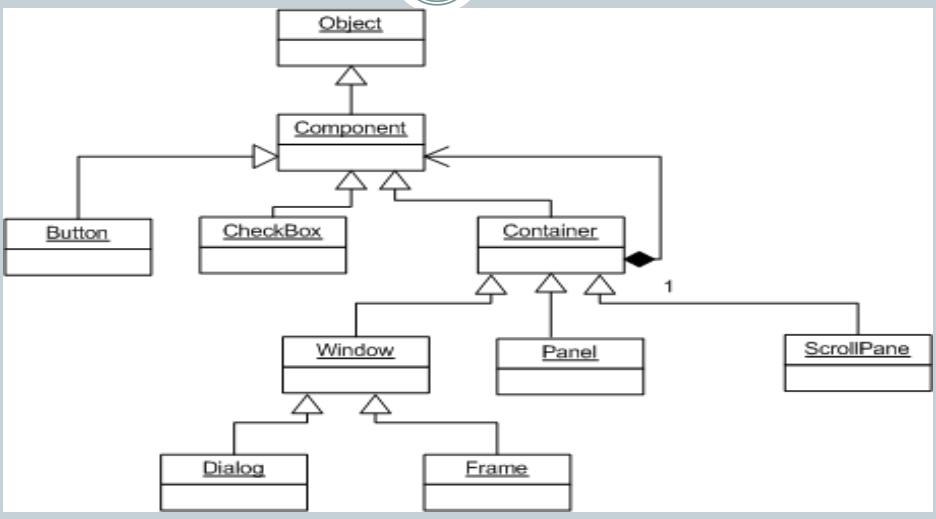
44



Composite Pattern

Example Java AWT

45



Composite Pattern

Steps to create a Composite Patte

46

- Ensure that your problem is about representing "whole-part" hierarchical relationships.
- Consider the heuristic, "Containers that contain containees, each of which could be a container." For example, "Assemblies that contain components, each of which could be an assembly." Divide your domain concepts into container classes, and containee classes.
- Create a "lowest common denominator" interface that makes your containers and containees interchangeable. It should specify the behavior that needs to be exercised uniformly across all containee and container objects.
- All container and containee classes declare an "is a" relationship to the interface.
- All container classes declare a one-to-many "has a" relationship to the interface.
- Container classes leverage polymorphism to delegate to their containee objects.
- Child management methods [e.g. addChild(), removeChild()] should normally be defined in the Composite class.

Flyweight

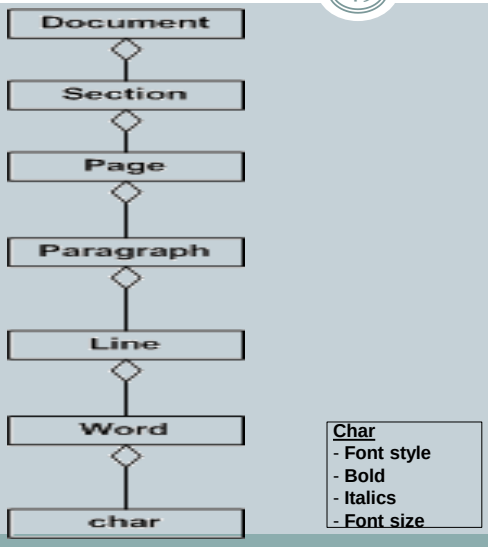
Flyweight

48

- Main purpose of Flyweight is to save space (memory)

Document Editor Problem Statement

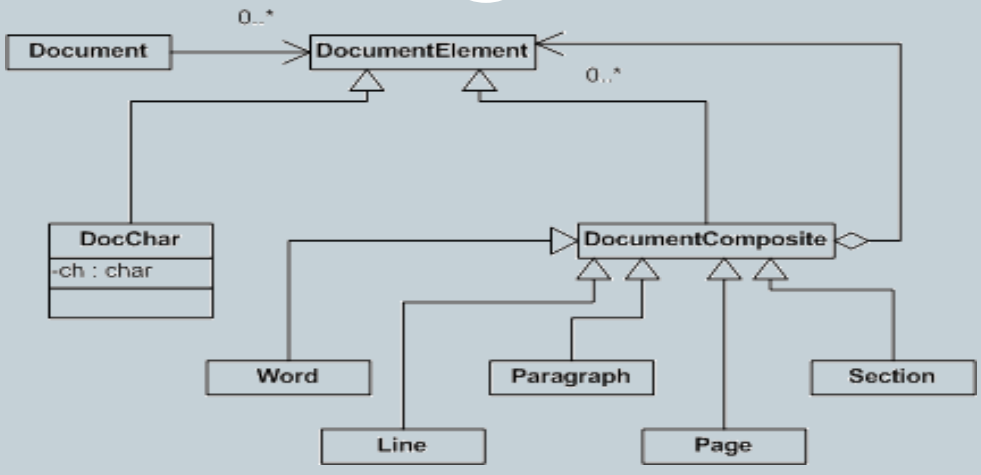
49



Flyweight Pattern

Using Composite

50



Flyweight Pattern

Issues with Composite approach?

- As a user types in some character in the document, a corresponding DocChar object is created and added to the document data structure.
- As a result large number of DocChar objects will be created in memory.

How much memory would be required ?

Consider 1 document ~ 4 sections ~ 25 pages each ~ Average of 5 paragraphs each ~ Average of 6 sentences in each ~ Average of 8 words in each ~ Average of 4 characters in each word.

We will need different variables to store each of them : (Considering 32 bit system)

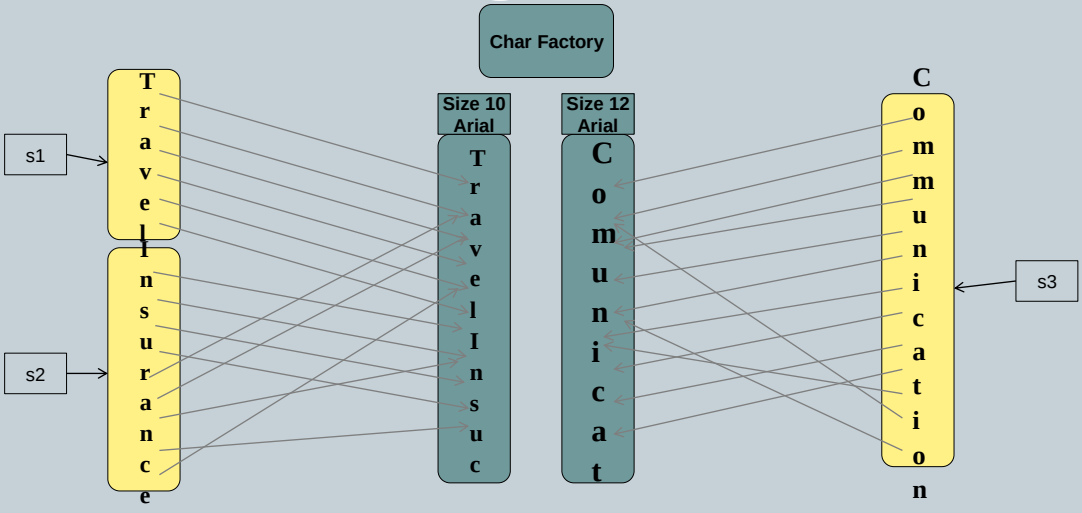
1 variable – **document**		1 x 4 bytes	= 4
4 variables – **section**		4 x 4 bytes	= 16
4 x 25 variables – **pages**	100 x 4 bytes	= 400	
100 x 5 variables – **paragraphs**	500 x 4 bytes	= 2000	
500 x 6 variables – **sentences**		3000 x 4 bytes	= 12,000
3000 x 8 variables – **words**		24,000 x 4 bytes	= 96,000
24,000 x 4 variables – **characters**		96,000 x 4 bytes	= 384,000
Total		123,605	494,420

Too much memory

53

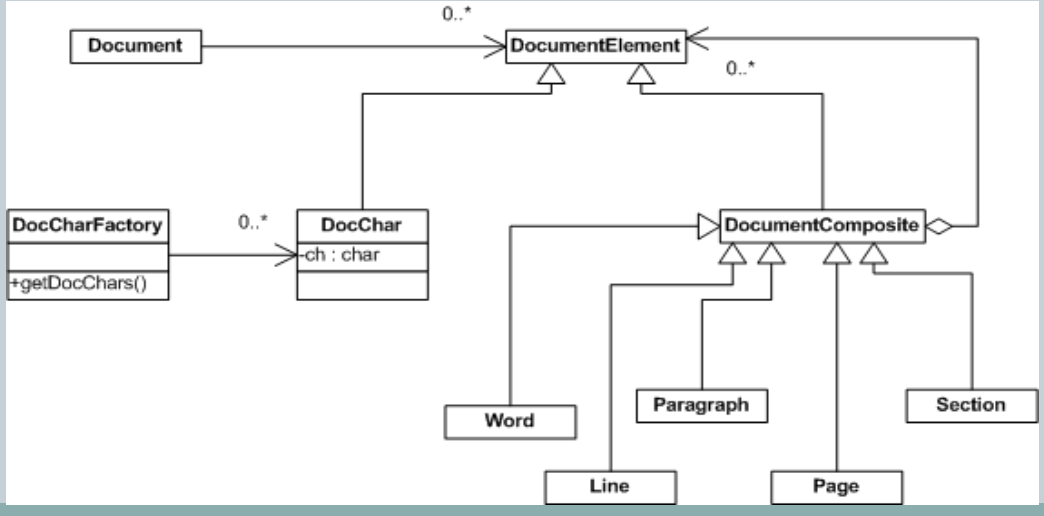
- How can we reduce the memory being utilized?
- Can we reuse something?
  - Lets start from the lowest element character
    - Do we need 96,000 variables for characters?
    - How many characters do I have (26 a-z, 26 A – Z, 0 – 9, special characters etc). In all around 60 usual characters. Considering each of them in different font sizes, styles used in the document might go up to 500 characters.
    - Can we reuse the 500 characters in different words instead of creating 96,000 characters in memory.

How can we reuse?



Flyweight

55



Flyweight Pattern

When to use Flyweight

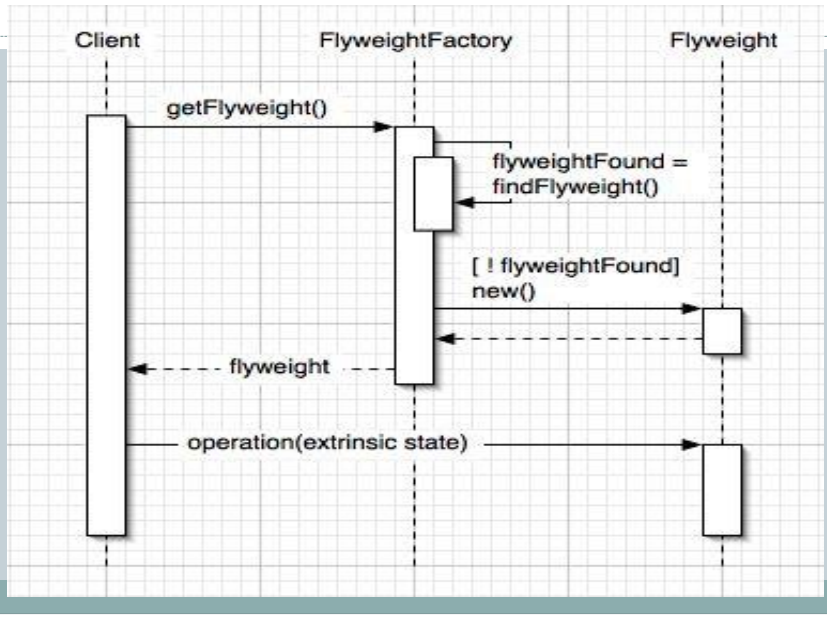
56

Use the Flyweight pattern when all the following are true :

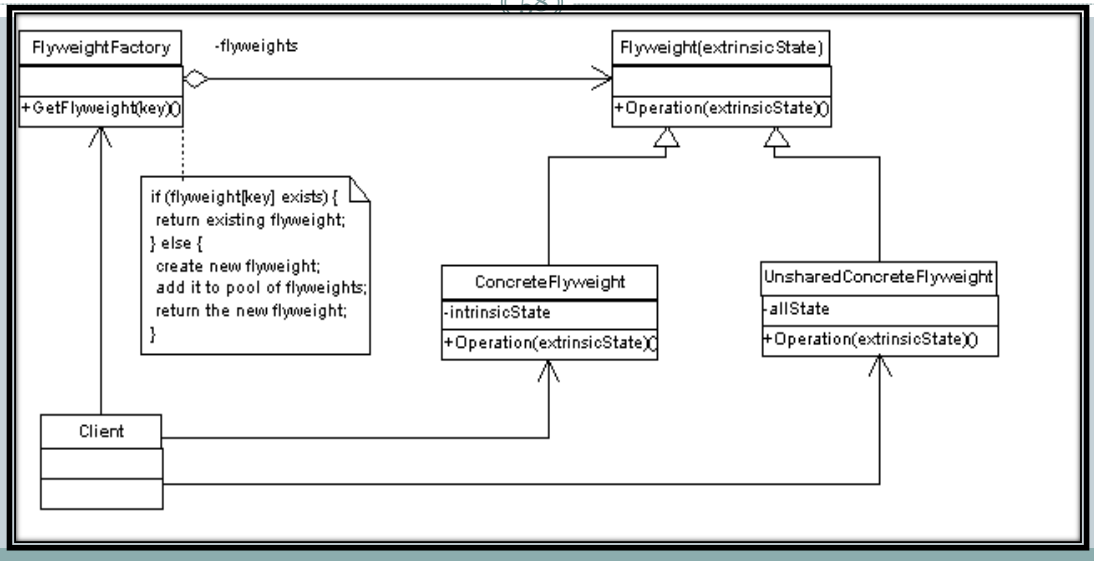
- An application uses a large number of objects
- Storage costs are high because of sheer quantity of objects
- Unique values of some object attributes is a small percentage of the total number of objects
- A large number of objects may, therefore, be replaced by relatively few shared objects
- The application does not depend on object identity. Since flyweight objects may be shared, identity tests will return true

Flyweight Pattern

Sequence Diagram



Structure



Flyweight Pattern

Benefits and Usage

Benefits

- Reduction in the number of objects to handle.
- Reduction in memory and storage devices if the objects are persisted.

Usage

- When the application uses large number of objects.
- Storage costs are high because of the quantity of objects.
- The application doesn't depend on object identity.

Proxy

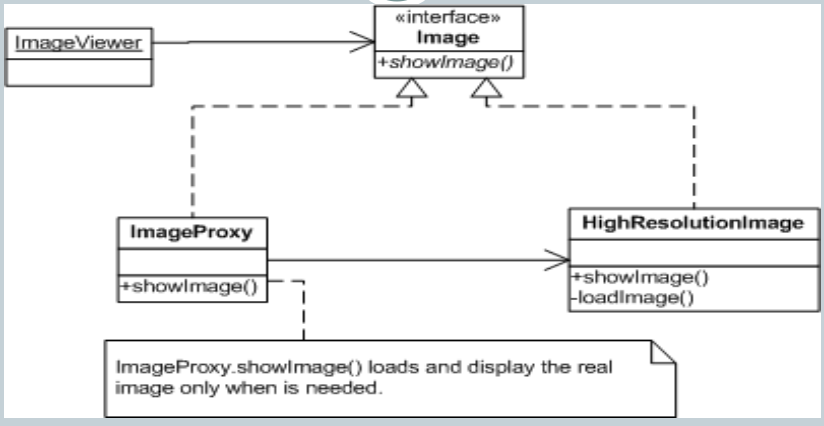
Proxy

61

- Proxy pattern is used to provide a surrogate or placeholder for another object to control access to it.
- One class controls the creation of and access to objects in another class

Proxy example

62



Proxy Examples

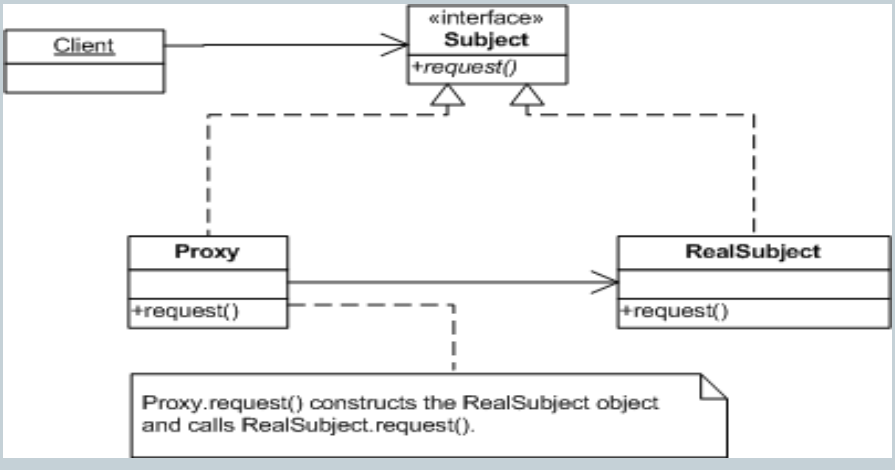
63

Below are some of the common examples in which the proxy pattern are used:

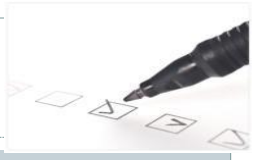
- Adding security access to an existing object. The proxy will determine if the client can access the object of interest.
- Simplifying the API of complex objects. The proxy can provide a simple API so that the client code does not have to deal with the complexity of the object of interest.
- Coordinating expensive operations on remote resources by asking the remote resources to start the operation as soon as possible before accessing the resources.
- Adding a thread-safe feature to an existing class without changing the existing class's code

Structure

64



Steps to implement proxy



65

- Identify the leverage or "aspect" that is best implemented as a wrapper or surrogate.
- Define an interface that will make the proxy and the original component interchangeable.
- Consider defining a Factory that can encapsulate the decision of whether a proxy or original object is desirable.
- The wrapper class holds a pointer to the real class and implements the interface.
- The pointer may be initialized at construction, or on first use.
- Each wrapper method contributes its leverage, and delegates to the wrappee object.

Proxy Pattern

```
namespace ProxyPatternDemo
{
    public interface IShape
    {
        string GetShape();
    }

    public class RealPolygon : IShape
    {
        public void Details()
        {
            Console.WriteLine("This is real polygon Class");
        }
        public string GetShape()
        {
            return "This is polygon shape from real/ actual class";
        }
    }

    public class ProxyPolygon : IShape
    {
        IShape _shape;
        public void Details()
        {
            Console.WriteLine("This is Proxy polygon Class");
        }
        public string GetShape()
        {
            _shape = new RealPolygon();
            return _shape.GetShape();
        }
    }
```

```
    class Program
    {
        static void Main(string[] args)
        {
            ProxyPolygon proxyClass = new ProxyPolygon();
            proxyClass.Details();
            string RealPolygonDetails = proxyClass.GetShape();
            Console.WriteLine(RealPolygonDetails);
            Console.ReadLine();
        }
    }
}
```

```
This is Proxy polygon Class
This is polygon shape from real/ actual class
```

66

Proxy implementations in Java

- java.lang.reflect.Proxy
- java.rmi.* (Remote Proxy)

Types of Proxies

There are four types of proxies, all taking the same basic format:

1. Virtual Proxy - The proxy won't create an "expensive" subject object until it is actually needed.
2. Remote Proxy - A local proxy object controls access to a remote subject object.
3. Protection proxy - The proxy insures that the object creating/calling the subject has authorization to do so.
4. Smart reference - The proxy will perform "additional actions" when the subject is called.

Benefits & Usage

Benefits

- A remote proxy can hide the fact that an object resides in a different address space.
- A virtual proxy can perform optimizations such as creating an object on demand.
- both protection proxies and smart references allow additional housekeeping tasks when an object is accessed.

Usage

- when you need a more versatile or sophisticated reference to an object than a simple pointer.

End of Chapter (Structural Patterns)